1 High-level Overview

This is the list of things we want to demonstrate with this system. Each has a title that describes the item best. Some are faults, some are data monitoring activities, and some are critical sequences for ensuring good data taking. Keep in mind that some of these are sort of typical examples of what one would do and that the processing concepts and methodologies are what is important.

1.1 Fault Related

1.1.1 Death of Filter Application

One of the filter programs running on an L3 worker node dies due to an exception condition such as "segmentation violation" or "illegal access". A logic problem in the application is likely to be the cause of the failture, activated by the prescence of unexpected event data configuration or size.

The filter program must be restarted with the same configuration and events must once again flow through it. The event that caused the problem will likely cause the application to fail yet again if it is fed back through. Actions that must take place:

* restart application
* save data that caused failure
* retrieve stack information of where the program was when failure occurred.
* send "filter died" message with stack information and event number out.

1.1.2 Disk Failure

Disk failures can be so bad that the machine is unuseable. Here, I believe, we are interested in a recognizable degradation in performance caused by a disk that is on its way to complete failure. Often the signal is too many bad sector reads/writes over some time period or reduced throughput.

When a disk is deemed too bad for operation, here is the list of things that must occur:

* send out notification that a problem has been found, two responses are possible:
  * automatic: start shutdown procedure now
  * notify operators: wait for a signal to leave or take it out of service
* try to run disk utilities that isolate the trouble spot
* ship important data off the machine
* take the machine out of service
* notify mainenance

1.1.3 CPU time too high

The applications running on the workers are using more time than the configuration predicted. This can be a local problem, a regional problem, or a global problem. This condition could mean a problem in the application or in the configuration or in one of the detectors.

Right now this can appear as an alarm condition in the control room. Later we can define some automatic recovery mode associated with this, such as growing the current partition using nodes reserved for offline use. Another is to check if the problem is localized at all - a handful of nodes on one switch.

1.1.4 Suspected Memory Leak

If we observe that the filter application's memory use is changing upwards with respect to time, we may conclude that the application has a memory leak.

Here is what we need to do:

* restart the application between events with same configuration

* report the failure and estimated memory loss per time period

Variation: It is likely that many of the filter app instances will experience the same problem and this will be seen everywhere continuously. A more subtle failure is that the memory is only leaked when certain types of events are spotted (e.g. b events) because addition code is executed or addition space is used in data structures. With this is mind, it MIGHT be possible to correlate event type with memory leak. If this is conceivable, then capturing one event that may be the type that causes the memory leak will be extremely valuable.

In the case that memory is creaping up everywhere, and the application is capable of reporting memory increases between functional units within itself, we should activate this feature on a select subset of nodes and gather memory increase numbers according to filter application contained function and report results. It is likely that such a facility, when activated, will degrade performance.

1.1.5 Machine disappears from farm

This one should be simple - detect that a node cannot be accessed.

1.2 Simple Monitoring Related

1.2.1 Raw Data Size versus CPU Processing time

The filter app internally is split up into paths and stages cooresponding to the processing of data for each of the subdetectors, combining that data, and making decisions about the data. Since we have raw data size info for each subdetector, and have measured the time per stage/path, we can produce a scatterplot of filter app stage/path CPU processing time versus raw data size. Even better is to match subdetector raw data size info with its matching stages. We can see if the size correlations and processing time actually make sense and are good. Such a measurement could also be used to locate bad spots in the code.

This particular item should be developed in a staged fashion:

1. for each event, on each participating node, send a message of (event_id,event_size) and another for (event_id,processing_time). Catch the messages at the monitoring GUI and produce points on the scatterplot

1. on each node, keep a mean/sd for a group of events and send the summary numbers up to the GUI app.

1. on each node, collect a small histogram of size vs time and send that summary up to the GUI app.

This type of exercise will show how an idea of what needs to be measured can be evolved from a simple brute force solution into something better.

1.2.2 Speed of filter diagnostic

The kernels we use should be outfitted so that any process on the ystem can be profiled by another process (e.g. a VLA or custom ARMOR element). Simple PC histogram profiling will be good enough. An operator should be able to indicate that he wants a 5 minute performance profile of the filter application. The system should respond by choosing one or more nodes to activate this feature on, collect the results, and make them available.

1.3 Physics information related

1.3.1 Hot/Dead Channel determination

A hot channel is one that is on all the time. A dead channel is one that seems never to produce data. The filter app can be configured to attempt to locate these. In the simplest form, this information is collected per channel (a histogram). All nodes should have this capability, but only a subset of nodes would have this feature activated.

* all nodes should have the capability, because we should try to keep all nodes "the same" for the sake of coherency and confidence in testing.

* only a subset would be active, because the "cost" of collecting the histograms is "large" compared to the value of identifying the hot and dead channels. We can not afford to have every node collect; we can not afford to have no node collect. Therefore, we need a subset. This is pure economics.

The collected data from the worker will be sent up to a management node, where is will be coalesced and sent up further until it reaches the top. The top will save it in a database to be available for subsequence runs or run segments.

1.3.2 Subdetector occupancy measurements

This is similar to the hot/dead channels. For each subdetector, a histogram of number of channels hit for each event will be collected. At some interval, the histograms will be release to an higher-level manager (a message). This manage will need to collect them and add the results together before shipping

results further upstream to the next level manager. At the top, the histogram will be shipped out of the trigger for display and storage.

1.3.3 Calibration/Alignment calculation

This is also similar to the previous two. After a certain number of events pass through the system, it will be possible to determine more precisely where individual sensors of the detector actually are. Precise run-time surveying of the detector (using the data itself) will help reduce systematic errors in the physics analysis, and will also help spot things that are "changing" over time. Targeted nodes will make these measurements and send them up at the end of a time interval or on demand. Once again, these calculated positions will need to be merged with measurement from other nodes, and finally stored for use in a database. The difference here is that after a new measurement is available, it must be propogated back to all nodes and be made ready-to-use. After all nodes have it ready, there will be an announcement made, through the run control mechanism, that a new run segment will start on event number X using this new alignment measurement. All events prior to X will use the old alignment, all ones after X will use the new one.

2 Commonality

Each one of the above scenarios has a similar appearance and similar characteristics.

Each one must be separated into entities that produces the data to be looked at, entities that look for anomolies or symptoms in the data, and entities that react to the situation.

{{{

data_source -> detection_1 -> reaction_1a -> reaction_1b -> ...

-> detection_2 -> reaction_2a -> reaction_2b -> ...

}}}

For this demo project, we can establish some rules regarding these sequences:

1. The sequences are determined at startup/configuration time.

1. Whether or not an item in the sequence or the sequence itself is active is a runtime decision.

1. each entity must be loosly coupled using elvin or ARMOR messages.

The processing scenarios (configuration and other info) are constructed at

design/development time and all relevant information in stored into a database.

2.1 configuration time

Having processing sequences available in the control room implies that no "make" or product builds will be done during configuration and startup.

2.2 run time changes

Whether or not an entity is actively doing work during runtime is controlled by an "authority vector".

2.3 loose coupling

Use of the messaging system implies that each part is * a VLA/custom_element/RC_statemachine using Elvin messages

* an Element using ARMOR messages

3 Odds and Ends

Each of the elements, vlas, filter apps, utility executables is going to need a parameter set when it starts. Any kind of thresholds or limits should some from the configuration database.

4 Uniqueness

Each of the scenarios described above has unique characteristics. This section is supposed to explain design/implementation details and strategy for each of the items.

Unfortunately I must stop typing for today and cannot get finished right now. Here are some ideas.

4.1 Death of filter app

Execution ARMOR restart app - this is the easy part. A signal handler should be activated before death. This handler could be part of the application or part of an external application that gets the "death of child" signal. This functionality can be viewed as being VLA-like and its job is to:

* get the call stack in address form

* get the memory usage information in /proc for this process

* use ELF information to resolve the stack address into function names

* [Mike: should this be done "deep" by the ARMOR, or done near the top

in the Run Control or Monitor?]

* send an elvin message out with all this information